Sierra IO System

Gregory D. Sjaardema
Advanced Computational Mechanics Architectures
Sandia National Laboratories

May 6, 2010

Contents

1 Introduction

The documentation below is a medium- to low-level view of the Sierra IO system targeted at developers who will be adding or modifying the database IO portion of the system. It should give enough detail that a new database type could be added by reading this document and looking at

an existing database class. It is also helpful to have the doxygen-generated documentation for the Ioss class hierarchy available.
The IO Subsystem has been designed to support multiple database formats simultaneously. It is possible to have the finite element model read from an ExodusII database; two results files being written to an ExodusII file with a third results file being written to an XDMF file; and the restart file being written to yet another ExodusII file. Each of these output databases can have a different schedule for when to write and what data is to be written.
If there are any questions or corrections, please contact Greg Sjaardema at (505) 844-2701 or gdsjaar@sandia.gov.

2 Class Structure Overview

The three main modules or namespaces involved in the IO are:

`Fmwk` Main sierra framework namespace. This defines and manages the Sierra applications data structures, algorithms, mechanics, etc. Each `Fmwk::Region` owns a `Frio::IOBroker` which is its link to all bulk data io.

`Frio` The bridge between the Sierra frameworks data structures and the IO databases data structures in the Ioss namespace. Ideally all data transfers between Sierra and the IO databases is under the control of the `Frio` classes. However, there are some special cases where an application will directly interact with the Ioss classes. The `Frio::IOBroker` manages the Restart, Mesh, Results, and other databases used by a `Fmwk::Region.`

`Ioss` The `Ioss` namespace contains several classes which attempt to provide an abstract interface to multiple concrete database types. It also defines a lightweight generic representation of the finite element model.

2.1 The Ioss Module

There are two major pieces of the `Ioss` module: The `Ioss::DatabaseIO` which is a pure virtual base class from which all concrete IO databases are derived and the `Ioss::GroupingEntity` classes which define a lightweight generic model representation.

The current inheritance structure of `Ioss::DatabaseIO` is shown in Figure 1. Currently the concrete

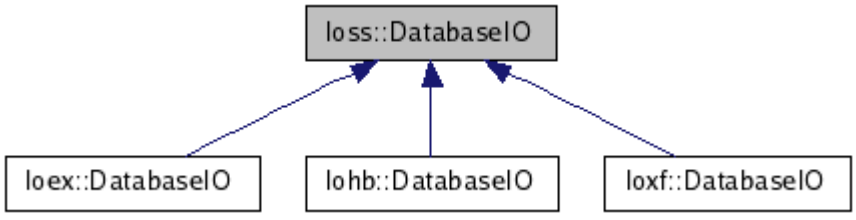

Figure 1: `Ioss::DatabaseIO` Inheritance Diagram

database types are ExodusII (`Ioex`), XDMF (`Ioxf`), and Heartbeat (`Iohb`). The `Ioss::DatabaseIO` class has a pointer to an `Ioss::Region` which is the root of the generic model representation.

2.1.1 Ioss::Region and Ioss::GroupingEntity Classes

An `Ioss::Region` is an `Ioss:GroupingEntity` and it contains a vector of `NodeBlocks`, `ElementBlocks`, `FaceSets`, `EdgeSets`, `CommSets` defining the current model. These are all also `Ioss::GroupingEntities`. The `Ioss::GroupingEntity` inheritance structure is shown in Figure 2. An `Ioss::GroupingEntity`

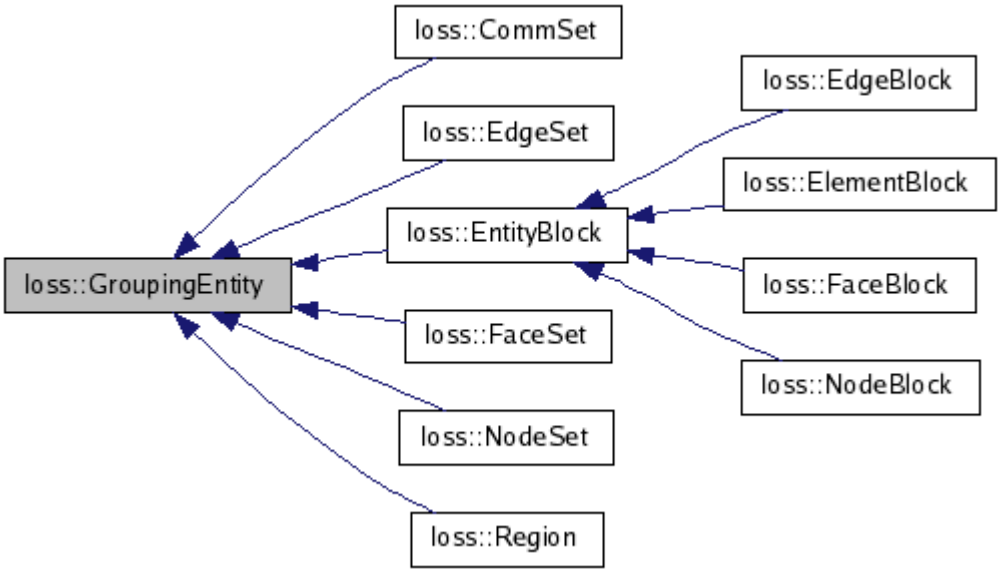

Figure 2: `Ioss::GroupingEntity` Inheritance Diagram

contains an `Ioss::PropertyManager` which maintains 1 or more `Ioss::Property;` and an `Ioss::FieldManager` which maintains 1 or more `Ioss::Field`. It also contains a name and a pointer to the `Ioss::DatabaseIO`. There is also an `Ioss::EntityBlock` which contains 1 extra member which is the `Ioel::ElementTopology` which is the topology of the entities contained in this block. The collaboration diagram for the `Ioss::GroupingEntity` is shown in Figure 3

A `GroupingEntity` is a very lightweight class; it does not contain any bulk data. It represents a portion of the finite element model and can transfer bulk data to/from the database from/to the Sierra datastructures. It also contains metadata about that portion of the finite element model. Each specific `GroupingEntity` type has a few required properties and fields and some optional properties and fields.

The `Ioss::GroupingEntity`'s define the basic finite element model. For example, a model with two element blocks, a nodeset, and two sidesets (facesets) is shown in Figure 4.

Note that the `FaceSets` are further divided into one or more `FaceBlocks` which contain entities of homogenous topology.

Each of these `GroupingEntity's` contain both `Properties` and `Field` definitions which can be queried by the `Frio` classes to define the model.

2.1.2 Ioss::Property and Ioss::PropertyManager

Each `Ioss::GroupingEntity` class contains an `Ioss::PropertyManager` which manages the `Ioss::Property` on this particular entity. An `Ioss::Property` is basically a named integer, real, string, or pointer

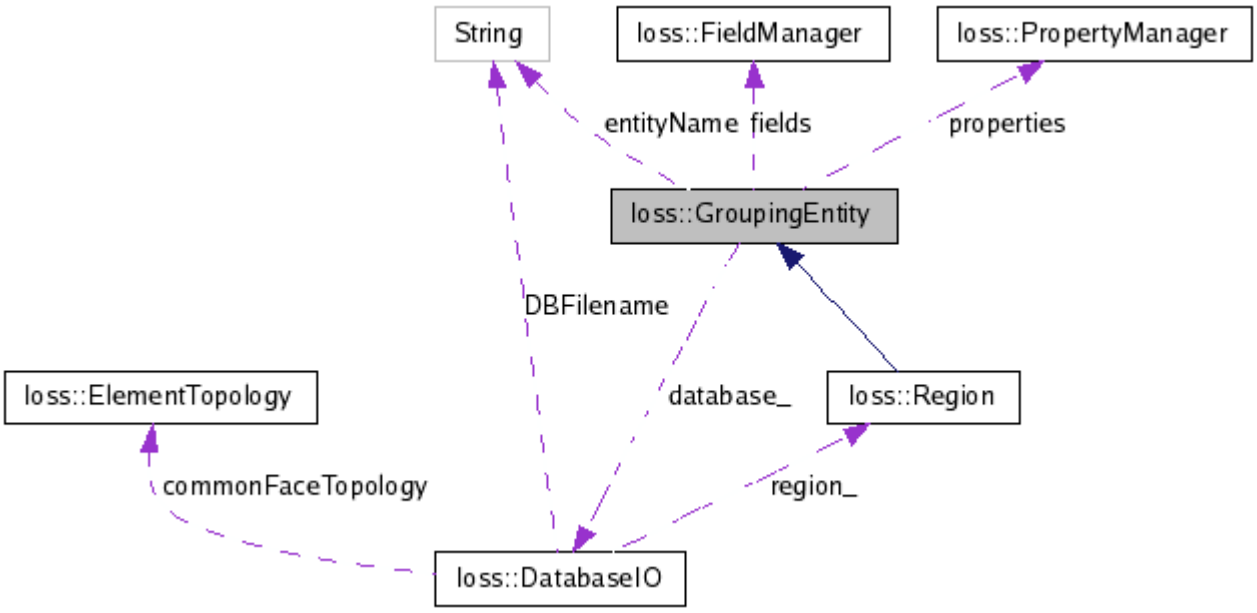

Figure 3: `Ioss::GroupingEntity` Collaboration Diagram

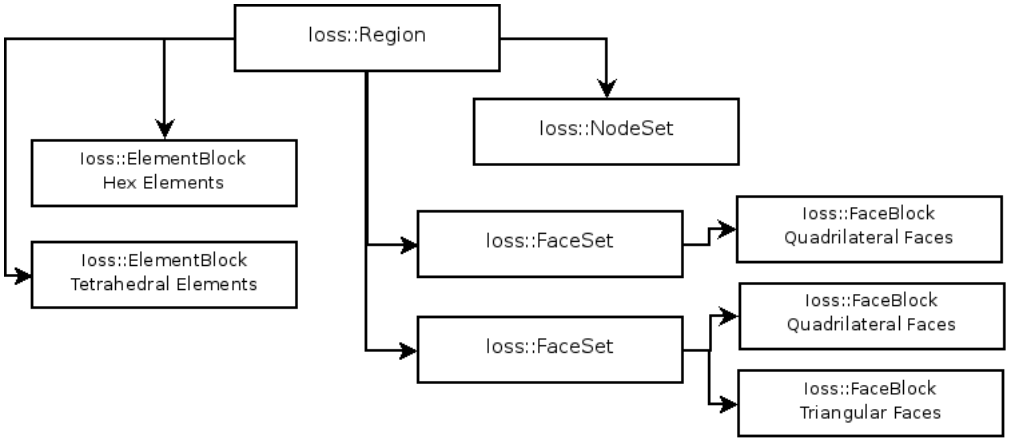

Figure 4: Block Diagram of Finite Element Model Structure

value. The PropertyManager maintains a list of all Properties for a specific entity and provides an interface where the Properties can be located by name.

2.1.3 Ioss::Field and Ioss::FieldManager

Each Ioss::GroupingEntity class contains an Ioss::FieldManager which manages the Ioss::Fields on this particular entity. An Ioss::Field contains the metadata for the field it describes; it does not hold the bulk data, so an Ioss::Field is fairly lightweight. Each Ioss::Field contains:

- std::string name – The name of the field.
- Int rawCount_ – The size of the untransformed field
- Int transCount_ – The size of the transformed field.
- BasicType type_ – The basic type of the field (Integer, Real, String)
- const VariableType * rawStorage_ – The storage type of the untransformed field (Vector, Scalar, Tensor, ...)
- const VariableType * transStorage_ – The storage type of the transformed field (Vector, Scalar, Tensor, ...)
- RoleType role_ – The "role" of the field. Valid roles are: INTERNAL, MESH, ATTRIBUTE, COMMUNICATION, INFORMATION, REDUCTION, and TRANSIENT. The most common are MESH for describing the model geometry/topology, ATTRIBUTE for block entity attributes (typically ElementBlock at this time), TRANSIENT for time-dependent results data, and REDUCTION for "summary fields" (see below).
- Int size_ – The number of bytes required to store the entire Field. Equal to rawCount_ * sizeof(type_) *
- std::vector< Iotr::Transform * > ransforms_ – A list of transforms which are applied to this field. *Note that the transforms are implemented at the Field level, but are not yet functional in Sierra applications.*

The Ioss::Fields are used for all bulk data input and output. Each Ioss::GroupingEntity contains a get_field_data and a put_field_data function. The function signatures are:

```
int get_field_data(const std::string &field_name, void *data, size_t data_size=0)
int put_field_data(const std::string &field_name, void *data, size_t data_size=0)
```

The functions will get or put the data corresponding to the field named field_name; the data will be read from/written to the memory pointed to by data which is of size data_size. If data_size is nonzero, then the field checks that there is sufficient space to store the field data; if it is zero, then it is assumed to be large enough. The function returns the number of entities for which the field was read/written.

The get_field_data and put_field_data functions are actually at the Ioss::GroupingEntity level and they forward the call down to internal_get_field_data and internal_put_field_data which are defined on each class derived from Ioss::GroupingEntity. These in turn call[1]:

[1] The offset and count were originally intended to provide partial field input and output, but this became too unwieldy and is not used at this time. Only full field input or output is supported.

```
int get_database()->get_field(this, field, offset, count, data, data_size);
```

get the concrete database class which is responsible for transferring the bulk data to/from the database to/from the memory pointed to by `data` of size `data_size` bytes.

Ioss::Field – Roles As defined above, there are six values for a fields `role`. The three most commonly used are:

`MESH` A field which is used to define the basic geometry or topology of the model and is not normally transient in nature. Examples would be element connectivity or nodal coordinates.

`ATTRIBUTE` A field which is used to define an attribute on an `EntityBlock` derived class. Examples would be thickness of the elements in a shell element block or the radius of particles in a particle element block.

`TRANSIENT` A field which is typically calculated at multiple steps or times in an analysis. These are typically "results" data. Examples would be nodal displacement or element stress.

`REDUCTION` A field which is typically summarizes some transient data about an entity. The size of this field is typically not proportionate to the number of entities in a `GroupingEntity`. An example would be average displacement over a group of nodes or the kinetic energy of a model. This data is also transient.

Ioss::Field – Storage The field storage members define the higher-order type of the variable such as vector, tensor, etc. The storage class is represented by the `Ioss::VariableType` base class which has several derived classes. The four types of derived class are:

- `ConstructredVariableType` – Specify a name and the number of components. This is typically used for user-defined storage types such as state variables. A specific variable may have 86 components, so the user would create composite variable type specifying a name and 86 components. By default, the IO system will create the type as `Real[component_count]`, so the above type would be known as "`Real[86]`".
- `CompositeVariableType` – A combination (or "composite") of the other field storage classes. For example, a field defined on the four integration points of a quadrilateral element would be a composite of 4 tensors. Currently the composite can only contain two subfields.
- `Ioel::ElementVariableType` – Each element known to the IO system will register a type which is sufficient for storing its nodal connectivity. For example, a 20-node hex registers the type `Ioel::St_Hex20` which has the name "hex20" and consists of 20 components.
- Predefined "common" types. This category encompasses many predefined types such as:

Scalar:	
Vector:	2D, 3D
Quaternion:	2D, 3D
Full_Tensor:	36, 32, 22, 16, 12
Sym_Tensor:	33, 31, 21, 13, 11, 10
Asym_Tensor:	03, 02, 01

Each variable type will return its component count. It also will return an optional suffix for each component. For example, the `Sym_Tensor_33` will return the suffices XX, YY, ZZ, XY, YZ, ZX.

2.1.4 Defined Properties and Fields

All classes derived from `GroupingEntity` provide the property:

`name`	(String)

In addition, the `NodeSet` class provides the properties:

`entity_count`	(Integer)
`distribution_factor_count`	(Integer)

And the fields:

`ids`	(Integer)	scalar
`distribution_factors`	(Real)	scalar

The `EdgeSet` and `FaceSet` classes provides the properties:

`block_count`	(Integer)
`edge_block_count`	(Integer) (for EdgeSet)
`face_block_count`	(Integer) (for FaceSet)

And no additional fields.

All classes derived from `EntityBlock` provide the properties:

`name`	(String)
`entity_count`	(Integer)
`topology_node_count`	(Integer)
`topology_type`	(String)
`parent_topology_type`	(String)

And the fields:

`ids`	(Integer)	scalar
`connectivity`	(Integer)	topology_node_count

The `NodeBlock` class provides the additional properties:

`component_degree`	(Integer)

And the additional fields:

`mesh_model_coordinates`	(Real)	component_degree

The `ElementBlock` class provides the additional properties:

`attribute_count` (Integer)

If `attribute_count` is greater than zero, then there will be additional fields defined for the attributes. These fields will all have the role `Ioss::Field::ATTRIBUTE`. In addition to the individual attribute fields, there will be a single Real field named "attribute" which contains a field for each element in the element block; the field will have `attribute_count` components.

The `EdgeBlock` and `FaceBlock` classes provides the additional properties:

`distribution_factor_count` (Integer)

And the field:

`element_side` (Integer) 2 (first component is element global id, second is local element ordinal; 1-bas

An `Ioss::Region` is an `Ioss::GroupingEntity`. In addition to the `name` property, it also provides properties that describe the overall structure of the model:

`entity_count`	(Integer)
`node_block_count`	(Integer)
`node_block_count`	(Integer)
`element_block_count`	(Integer)
`face_set_count`	(Integer)
`edge_set_count`	(Integer)
`node_set_count`	(Integer)
`comm_set_count`	(Integer)
`node_count`	(Integer)
`element_count`	(Integer)
`state_count`	(Integer)
`current_state`	(Integer)
`database_name`	(String)

2.2 Ioss::DatabaseIO Class Hierarchy

The `Ioss::DatabaseIO` class is a virtual base class which defines the interface required of each concrete database format. In addition, it also provides some functionality which is common to all (or many) database types.

2.2.1 Attributes

The `Ioss::DatabaseIO` class has the following attributes:

`std::string DBFilename` –The name of the file this database is reading or writing.

`Ioss::State dbState` –An input database will always be in `STATE_READONLY` which signifies that it cannot be written to or modified. An output database follows a set order of access. The states corresponding to this access are:

STATE_INVALID	Error state if something goes wrong.
STATE_UNKNOWN	Typically used at the very beginning of the databases existence when the class has been created, but no reading or writing has occurred.
STATE_READONLY	An input database is only in either STATE_UNKNOWN or in STATE_READONLY which means that it cannot be written to or changed.
STATE_CLOSED	The states are not nested, so each state must end with a transition to the STATE_CLOSED prior to entering the next state.
STATE_DEFINE_MODEL	Defining the metadata which defines the model (non-transient, geometry and topology).
STATE_MODEL	Outputting the bulk data (mesh_model_coordinates, ids, connectivity) relating to the model portion.
STATE_DEFINE_TRANSIENT	Defining the metadata relating to the transient data. For example, the element or nodal fields.
STATE_TRANSIENT	Outputting the transient bulk data.

`bool isParallel` – true if running in parallel

`int myProcessor` – the processor this instance is running on.

`TopoContainer faceTopology` – There are times when a Sierra application needs to know the topology of the faces in the model. This container contains a list of the face topology types in the model. It is populated by the concrete database types.

`Ioel::ElementTopology *commonFaceTopology` – If there is a single face topology in the model, this is set to that value; otherwise it is `NULL`. This is used to speed up some face topology queries.

2.2.2 Interface

Each concrete database class must derive from the `Ioss::DatabaseIO` class and provide an implementation of several functions. The most obvious of these is the constructor:

```
DatabaseIO (Region *region, const std::string &filename, Ioss::EventInterest db_usage)
```

The constructor is passed a pointer to an `Ioss::Region`, a `filename`, and a '`db_usage`' field. The region can be `NULL` at this time if the database is an input database. The `db_usage` specifies what this database will be used for and have valid values of `WRITE_RESTART`, `READ_RESTART`, `WRITE_RESULTS`, `READ_MODEL`, `WRITE_HISTORY`, or `WRITE_HEARTBEAT`. If this is an output database, then it should not be opened at this time since the user may have specified the same file for the reading of an initial state or a restart and we don't want to overwrite any data that may be needed. If an input database, then it can be opened to check that it exists, but no major processing of data should occur.

For an input database, data processing will occur in the '`read_meta_data()`' function. When this is called, the database will be in state `STATE_DEFINE_MODEL`. It should then create all `GroupingEntities` and add them to the `Region` to create the lightweight `Ioss` model. Any properties should be added

to the GroupingEntities at this time also as should transient fields. Basically, the Ioss system should have a complete metadata representation of the model at this time and should be able to respond to any queries about the model that do not involve bulk data without accessing the database.

For an output database, the Sierra application through the Frio classes will build the Ioss metadata representation of the model while in STATE_DEFINE_MODEL. When Ioss::DatabaseIO::end() is called to exit from STATE_DEFINE_MODEL, then the metadata model is complete and the database can write whatever non-bulk data is needed at that point.

The Ioss system will next go into the STATE_MODEL at this point and the fields with a role of MODEL will be written or read via the field interface.

As documented earlier, all bulk data reading and writing is done via Ioss::Fields. The Ioss::DatabaseIO class has several field input and output functions; one for each GroupingEntity type:

```
int  get_field (const Region *reg, const Field &field, void *data, size_t data_size)
int  get_field (const NodeBlock *nb, const Field &field, void *data, size_t data_size)
int  get_field (const ElementBlock *eb, const Field &field, void *data, size_t data_size)
int  get_field (const FaceBlock *fb, const Field &field, void *data, size_t data_size)
int  get_field (const EdgeBlock *fb, const Field &field, void *data, size_t data_size)
int  get_field (const NodeSet *ns, const Field &field, void *data, size_t data_size)
int  get_field (const EdgeSet *es, const Field &field, void *data, size_t data_size)
int  get_field (const FaceSet *fs, const Field &field, void *data, size_t data_size)
int  get_field (const CommSet *cs, const Field &field, void *data, size_t data_size)

int  put_field (const Region *reg, const Field &field, void *data, size_t data_size)
int  put_field (const NodeBlock *nb, const Field &field, void *data, size_t data_size)
int  put_field (const ElementBlock *eb, const Field &field, void *data, size_t data_size)
int  put_field (const FaceBlock *fb, const Field &field, void *data, size_t data_size)
int  put_field (const EdgeBlock *fb, const Field &field, void *data, size_t data_size)
int  put_field (const NodeSet *ns, const Field &field, void *data, size_t data_size)
int  put_field (const EdgeSet *es, const Field &field, void *data, size_t data_size)
int  put_field (const FaceSet *fs, const Field &field, void *data, size_t data_size)
int  put_field (const CommSet *cs, const Field &field, void *data, size_t data_size)
```

The implementation of these functions simply do some optional logging and then call the private pure virtual function put_field_internal(...) with the exact same arguments that they were called with.[2] The underlying concrete database must either read or write the specified field when the function is called. It is recommended to treat the 'data' field as readonly at this time, so if any reordering or processing of the data is required, a scratch array should be allocated to hold the modified data[3]. One of the first fields read/written from/to most GroupingEntities is the "ids" field. This defines a global id for each entity in the GroupingEntity and also defines the ordering of all subsequent fields on that entity. For example, if the id field for a nodeblock is {1,5,2,4,3}, then a nodal coordinate of {1.0, 2.0, 3.0, 4.0, 5.0} would assign 1.0 to node 1, 2.0 to node 5, 3.0 to node 2, 4.0 to node 4 and 5.0 to node 3. There is a caveat to this which is discussed in a later section.

[2] See http://www.gotw.ca/publications/mill18.htm for a discussion of why (pure) virtual functions should not appear in the public interface. Note that this guideline is both respected and violated in Ioss::DatabaseIO....

[3] I am considering adding an argument to the field functions which would indicate whether the 'data' can be modified or should be treated as readonly. This could help a little with memory use, but most cases don't need to do any modifications of the field....

Once the STATE_MODEL state is finished, the application may enter the STATE_DEFINE_TRANSIENT state for an output database. It will then define what fields are to be written to each of the GroupingEntities. Once these fields are defined, the database will enter STATE_TRANSIENT.

In STATE_TRANSIENT, the transient fields are written. The application will call:

```
Ioss::DatabaseIO::begin_state(Ioss::Region *region, int state, Real time)
```

which tells the database that the application will start writing transient fields applying to time 'time'. The 'state' will be an sequentially increasing counter which indicates how many transient steps have been written. It is 1-based. This function can also be called for an input database if the application is going to be reading transient data from the input database. In this case, access to the state can be essentially random; the application does not have to read from step 1...end_step sequentially. The function:

```
Ioss::DatabaseIO::end_state(Ioss::Region *region, int state, Real time)
```

will be called when all transient fields for this state have been written. At this time, the DatabaseIO class can do whatever is needed to finalize this particular timestep.

2.2.3 Database Traits Interface

In addition to the model and field-related functions defined above, there are a few functions which specify characteristics of the database. They are:

`virtual bool supports_nodal_fields()` – true/false depending on whether this database type supports fields written on nodes in nodeblocks.

`virtual bool supports_edge_fields()` – true/false depending on whether this database type supports fields written on edges in edgesets.

`virtual bool supports_face_fields()` – true/false depending on whether this database type supports fields written on faces in facesets.

`virtual bool supports_element_fields()` – true/false depending on whether this database type supports fields written on elements in element blocks.

`virtual bool supports_nodelist_fields()` – true/false depending on whether this database type supports fields written on nodes in nodelists.

`virtual Int node_global_to_local(Int global, bool must_exist)` – Provides a mapping from a global node id down to the local 1..entity_count position in the field data. In the 'id' and coordinate example above, the local position of global node '5' is '2'. If the boolean '`must_exist`' is true, then it is an error if the specified node does not exist; otherwise if `must_exist` is false, then the function returns '0' if a node with that global id does not exist.

`virtual int maximum_symbol_length()` – Return the maximum length of the 'field names' that the database can handle; return 0 if it is unlimited. This is used by the restart system which must 'mangle' the field names down to the maximum size that can be supported on the database.

3 Special Cases –Dynamic Topology

Although the above description is valid for most cases, there are some situations which must be handled to allow the database to be used in all situations. Dynamic topology which include element death, load balancing, and adaptivity cause the topology of the original mesh to change from the originally defined topology.

3.1 Element Death

In the element death case, elements are selectively "killed" or inactivated based on some criteria in the application. Most databases continue to output these elements but they are flagged via an element variable as whether they are active or inactive. This shouldn't cause much difficulty, but the Sierra framework reorders the elements when element death occurs, so the output database must map the element fields which are not ordered based on the new element ordering back to the original element ordering so that they can be output correctly. When the element id ordering is changed, the `Frio` system will inform the database by calling a `put_field` on the element block entities "ids" field with the new ordering. This can be distinguished from the original output of the element ids by the fact that in the original ordering, the database will be in `STATE_DEFINE_MODEL`; in any subsequent reordering of the element ids, the database will be in `STATE_TRANSIENT`. All subsequent output of element transient fields will then be in the current ordering and will need to be mapped to the original ordering before being written to the database.

3.2 Load Balance

The load balance case involves shuffling the current nodes and elements among processors. One or more nodes and elements will move to a new processor in this case. The current handling of this case is suboptimal and is based on the characteristics of the ExodusII database format which is the primary format currently used in Sierra applications. The ExodusII database creates a file per processor and all communications to these databases are independent of all other databases. At the end of the calculation, a separate program is used to join the per-processor databases into a single database. Because the per-processor databases are independent, the current handling of a load balance reshuffling is that the current databases are all closed and a new set is opened; a new model is defined; the transient fields are redefined; and then everything continues from that point.

This is definitely not the optimal situation and will be changing in the future. It is expected that a new function will be added to the `Ioss::DatabaseIO` traits interface which will indicate whether a concrete database instance supports dynamic topology. If it is supported, then the `Frio` system will notify the database that shuffling of elements and nodes among processors has occurred and output a new set of node, element, face, and edge orderings. If the database does not support dynamic topology, execution will continue as above.

3.3 Adaptivity

The adaptivity case is handled the same way as the above load balance case. The only difference is if the database is being used to store restart data. In that case, instead of just outputting fields existing on all of the current generation nodes and elements; fields are output on the complete

element and node hierarchy including the parent elements which have been adapted. As stated above, the current behavior will be changing in the future.

4 Database Registration

Before a database type can be used in a Sierra application, it must be registered and it must provide a method for creating an instance of itself. This is currently handled via an `Ioss::IOFactory` class as shown below:

```
namespace IoXX {

  const IOFactory* IOFactory::factory(){
    static IOFactory registerThis;
    return &registerThis;
  }

  IOFactory::IOFactory()
    : Ioss::IOFactory("database_type") {
    Ioss::IOFactory::alias("database_type", "alias_for_database_type");
  }

  Ioss::DatabaseIO* IOFactory::make_IO(const std::string& filename,
                                       Ioss::EventInterest db_usage) const
  { return new DatabaseIO(NULL, filename, db_usage); }
}
```

The database type must also be made known to the parsing system. This is done by editing the file `framework/parser/io/Ioss_MeshInput.xml` and adding an enum_entry with "`database_type`" to the enum `DatabaseTypes` near the top of the file.

Since Sierra applications are currently statically linked, some function in your database library must be called by an existing piece of Sierra code in order for the linker to pull in your library. If the database is to be available for all codes, the best way to do this is to add a call to your database's `IOFactory` to the code in `Ioinit::Initializer()` as shown below:

```
namespace Ioinit {
  Initializer::Initializer() {
    Ioex::IOFactory::factory();    // ExodusII
    Iohb::IOFactory::factory();   // HeartBeat
    IoXX::IOFactory::factory();  // My new database type
  }
 }
```

If the database is only available to a few applications, then the call to the `factory()` method is usually placed in the applications main routine. If a database has some finalization functions which must be called at the end of execution (this is for a library, not a specific database instance), then a call to that code can be added to the `Ioinit::Initializer` destructor.

The user can then request this database type in the input file using the line command Database Type = database_typ
or Database Type = alias_for_database_type

5 Build System

The `io_system` uses the standard Sierra build system which is currently `BJam`. The main documentation page for BJam is http://boost.org/boost-build2/index.html, and documentation for using BJam with Sierra applications can be reached from the Sierra developers documentation page at http://swi.sandia.gov/developers/develop.php. The build system must also be notified that the new library is to be built and linked to the other code. The building of the library is controlled in the ***NbtTools/io_system/votd/Jamfile*** file. If the new database requires any external libraries (which are referred to as "TPL's" which means "third-party libraries") which provide an API or other functionality, they are specified in the rules for the library. For example, the `ioex` library (exodusII) requires the exodusII, nemesis, and netcdf libraries and that is specified as:

```
lib ioex
  :
    [ glob $(io_system-root)/src/exodusII/*.C ]
    ioss
    /sierra/utility//utility
    /tpl/exodus//exodus
    /tpl/nemesis//nemesis
    /tpl/netcdf//netcdf
  ;
```

The specification of what library versions will be used is specfied in the ***Nbtools/Jamroot*** file:

```
...
  register-product tpl : exodus  : exodusii : 4.72 ;
  register-product tpl : nemesis : nemesis  : 3.09 ;
  register-product tpl : netcdf  : netcdf   : 3.6.2-snl1 ;
...
```

This specifies that version 4.72 of the ExodusII library, version 3.09 of the Nemesis library, and version 3.6.2-snl1 of the netcdf library will be used. Later on in the file, the Ioex library build instructions are defined as:

```
lib ioex
  :
    [ glob $(io_system-root)/src/exodusII/*.C ]
    ioss
    /sierra/utility//utility
    /tpl/exodus//exodus
    /tpl/nemesis//nemesis
    /tpl/netcdf//netcdf
  ;
```

This specifies that the Ioex library is to be built. The source code for the library is in ***src/exodusII***. And it consists of all ***.C*** files in that directory. The `Ioex` library "requires" the `ioss` library from the io_system; the external `utility` library from `/sierra/utility`; and the `exodus`, `nemesis`, and `netcdf` third-party-libraries (tpl). There are similar sections for the other libraries that are part of the io_system package.

One last piece of information is to tell the build system how to build an application that uses the IO libraries. The build system is told this in the following code block for the `io_shell` executable:

```
exe io_shell
  :
    $(io_system-root)/src/main/Main_io_shell.C
    ioinit
    ioss
    iotr
    ioxf
    /sierra/utility//utility
  : <tag>@runtest-tag
  ;
```

Using this information, the build system should be able to generate compile flags and correctly order the libraries on the link line for the linker. A new database type can be added by following the patterns for the Ioex, Ioxf, and Iohb database types.

6 Testing

The Sierra unit test system contains an executable called `Utst_io` which can be used to test the various `DatabaseIO` classes. This executable is part of the Sierra system, but it is also available in a standalone version which can be built outside of the Sierra tools. The executable reads a database of a specified format and then writes the data to a database of a specified format. This can be used to test your `DatabaseIO` class to make sure it reads the data correctly and writes the data correctly. It can also be used as a translator from one database format to another.

7 Summary

The above documentation gives a brief overview of the Sierra IO system. It is not enough detail for the reader to add a completely functioning new database type, but hopefully gives enough detail that the reader can understand existing database IO classes and using those as an example add a new database type.

There are some idiosyncrasies in the current Ioss system, but hopefully as they are discovered, they will be removed via refactoring. The current interface is definitely biased towards the ExodusII functionality, but as more database types are added, I will attempt to remove these biases.